StarShow当期03：BE CRASH疑难杂症的排查经验 - 冯浩桉

crash的几种原因:

BE OOM 引起的 Crash

这里的OOM有两种含义，一种是BE被操作系统直接kill了(dmesg中可以看到 kill process)，另一种是BE在申请内存的时候操作系统返回了NULL(`overcommit_memory` = 1)，我们这里是指第二种

在大多数情况下，我们的BE都是可以正确处理OOM的，但是对于一些极端的场景，我们的OOM处理相对还是比较困难的，(比如使用了一些第三方网络框架)。

如果我们的BE内存配置的是正确的，我们在大多数都不会被操作系统直接kill，而是会因为出发了我们的内部熔断逻辑把任务 cancel 掉，少部分可能是因为某些thread local逻辑没处理好

查询消耗大量内存导致OOM

我们的内存分配库中会记录内存申请使用量，在2.3以下的版本中，tcmalloc会把内存使用的堆栈dump出来(be.out)

```
1 tcmalloc: large alloc 3776905216 bytes == 0xf0f37c000 @  0x5819bbf 0x5aab21c 0x2
```

然后通过binutils把具体的行号打印出来:

```
1 addr2line -e lib/starrocks_be 0x5819bbf 0x5aab21c 0x20c1b08 0x59fb6b5
```

在2.4+之后，我们把内存分配库替换成了jemalloc，我们同样实现了这样的功能，但是打印的日志换成了 be.WARNING

```
1 W1226 01:46:59.816656 13960 mem_hook.cpp:266] large memory alloc: 3826657572 byt
2     @           0x3cb408b  malloc
3     @           0x6b48305  operator new()
4     @           0x6b483d9  operator new[]()
5     @           0x5e8c706  simdjson::haswell::dom_parser_implementation::set_capa
6     @           0x5e74e30  simdjson::haswell::implementation::create_dom_parser_i
```

```
 7     @          0x226e5ab  simdjson::fallback::ondemand::document_stream::start()
 8     @          0x226be5d  starrocks::JsonDocumentStreamParser::parse()
 9     @          0x2260010  starrocks::JsonReader::_read_and_parse_json()
10     @          0x2263a57  starrocks::JsonScanner::_open_next_reader()
11     @          0x2265440  starrocks::JsonScanner::get_next()
12     @          0x42fbf21  starrocks::connector::FileDataSource::get_next()
13     @          0x236336a  starrocks::ConnectorScanNode::_scanner_thread()
14     @          0x3bd8280  starrocks::PriorityThreadPool::work_thread()
15     @          0x4eeb187  thread_proxy
16     @     0x7efd98b7dea5  start_thread
17     @     0x7efd981988dd  __clone
18     @                (nil)  (unknown)
```

这样我们就可以在BE crash的时候通过这些大内存申请信息来确定是否是大内存消耗场景考虑的不周全导致BE crash。

> 所有报OOM的问题真的都是OOM问题吗?
>
> 不一定，在某些情况下，虽然会报OOM，但是实际上我们的内存使用没有这么多，可能是内存写越界导致的，也有可能是某些列的类型对不上导致某些列的长度拿到了一个负值，这也会导致上报OOM
>
> https://starrocks.atlassian.net/browse/SR-12047
>
> https://forum.mirrorship.cn/t/topic/3557

导入/Compaction消耗内存

我们在做导入或者是在做compaction的时候，如果没有控制好任务的并发控制，有可能导致我们的BE消耗的内存太多。导致了BE的crash了，这些crash的栈底一般也是跟compaction或者是导入相关的

查询引起的crash

查询是我们经常引起crash的一个原因。为了方便排查线上问题，我们总结了很多排查查询crash的手段

快速定位Crash SQL

1. 通过 be.out 中 crash 之后的 query_id/fragment_id 来定位SQL

原理: 我们把查询的query_id和fragment_id绑定到运行时的thread local中，这样在BE crash的时候就可以把这些变量dump到文件中了。然后我们就可以通过query_id来去fe的审计日志中找到对应SQL。这在大多数场景下都是能快速找到我们出问题的SQL。但是他还是有一定的局限性的。

```
1 query_id:41156738-81c7-11ed-ac7b-cae020930719, fragment_instance:41156738-81c7-1
```

```
tracker:tablet_metadata consumption: 1893125
tracker:rowset_metadata consumption: 1035389
tracker:segment_metadata consumption: 6552
tracker:column_metadata consumption: 40304
tracker:tablet_schema consumption: 34973
```

```
2022-12-22 15:07:11,515 [query]
|Client=127.0.0.1:50902|User=root|AuthorizedUser='root'@'%'|ResourceGroup=|Cata
log=default_catalog|Db=ssb|State=ERR|ErrorCode=THRIFT_RPC_ERROR|Time=640|ScanBy
tes=0|ScanRows=0|ReturnRows=0|StmtId=26|QueryId=41156738-81c7-11ed-ac7b-
cae020930719|IsQuery=true|feIp=172.26.92.227|Stmt=select p_partkey from part
order by p_size limit 10|Digest=|PlanCpuCost=80.0|PlanMemCost=80.0
```

因为我们只是把crash线程的query_id dump出来。但是我们执行线程可能是bthread运行的，如果发生yield了，这个query id可能是一个不相关的query-id。还有一种可能就是我们的query有可能是并发执行的。有可能是其他的query把内存写花了，但是当前的query仍然访问了这个内存，这也会导致我们crash的dump出来的query_id和真正原因的query id之间有很大差距。所以这个手段只能是作为一个辅助排查的工具，获取到这个SQL之后要尽量试一下是否真的是可以复现。(否则可能会引入很多无意义的工作量) 如果栈是比较稳定crash在某一个地方，那么还是大概率是这个SQL导致的

query_id 是 0x0000000 问题的原因:

- 可能不是查询引起的，目前只有查询 (包括insert into select) 才会设置这个query_id，如果是导入引起的那么就不会设置这个变量
- 在较低的版本下没有对 RPC/SCAN 线程设置query_id，2.4+之后基本上跟查询相关的线程都设置了query_id

be.out 在crash的时候没有任何的输出 (栈也没有):

- 检查是不是被操作系统kill了 (没有coredump，但是dmesg中有kill记录)
- 检查是不是stack overflow (有coredump，可以通过coredump进一步定位)

2. 通过 gdb runtime_state 中的 query_id/fragment_id 来定位SQL

如果我们有用户crash的coredump，就可以拿到很多的信息，包括 query_id 等。我们的执行线程中可能会有一些query_id信息，通常是在RuntimeState对象里面。我们只要在coredump中打印出来RuntimeState信息，就可以获取query_id或者是fragment_id，然后就可以找到具体的SQL了。

我们可以写一些debug脚本来帮助打这些变量

```
python
import uuid
import gdb

class getQueryId(gdb.Function):

    def __init__(self):
        super(self.__class__, self).__init__('get_query_id')

    def invoke(self, runtime_state):
        if str(runtime_state.type) != 'starrocks::RuntimeState *':
            return "not a runtime state"

        query_id = runtime_state.dereference()['_query_id']
        hi = int(query_id['hi'])
        lo = int(query_id['lo'])

        if hi < 0: hi =(1 << 64) + hi
        if lo < 0: lo = (1 << 64) + lo
        value = (hi << 64) + lo
        x = uuid.UUID(int = value)
        return str(x)

getQueryId()
end

call $get_query_id(_runtime_state)
$16 = "f9e96830-85c8-11ed-80e0-c602da2cf328"
```

定位Crash原因

1. 通过 SQL 来具体定位原因:

如果我们已经发现是某个SQL会引起BE的crash，那么我们就可以进一步分析Crash的原因了

某个SQL会引起crash，它既可能是因为优化器错误引起的，也可能是因为执行器错误引起的。

优化器引起的错误一般都是丢列，列类型对不上，nullable属性对不上。执行器的错误可能有空指针，常量/null处理的有问题。

为了加快问题具体原因的定位速度，我们最好是可以找一个最小复现case。

因为用户出问题的SQL可能是非常复杂的，有可能有几百几千行的SQL，直接看explain或者是复现是很困难的，为了缩小排查的范围，我们最好通过改写SQL来确定一个最小的复现case。比如用户的SQL可能是由多个SQL union在一起的。直接查某一个子查询都是没问题的，但是union在一起就会有问题，这个时候我们就要考虑是不是plan在处理union的时候有问题。

我们确定好一个最小复现case之后，就可以先看一下plan是否是有问题 (如果拿到了coredump，也可以优先去看某个算子附近的plan)

列丢失:

shuffle 的时候期望shuffle列647，但是project节点只会生成 324。这个时候会导致我们shuffle的时候丢列，可能造成在exchange或者是接收端在收发数据的时候 crash

```
  51:EXCHANGE
     cardinality: 3662648

PLAN FRAGMENT 3(F63)

  Input Partition: HASH_PARTITIONED: 647: case   324: date
  OutPut Partition: HASH_PARTITIONED: 647: case   324: date
  OutPut Exchange Id: 107

  106:Project
  |  output columns:
  |  324 <-> [324: date  DATE  true]
  |  hasNullableGenerateChild: true
  |  cardinality: 1009517
  |  column statistics:
  |  * date-->[1.6687008E9  1.67E+09  0 4  24.822695035460992] ESTIMATE
  |
  105:AGGREGATE (merge finalize)
  |  aggregate: sum[([648: sum  DECIMAL128(38  6)  true]); args: DECIMAL128; result: DECIMAL128
  |  group by: [647: case  VARCHAR(65533)  true]  [324: date  DATE  true]
  |  having: cast([648: sum  DECIMAL128(38  6)  true] as DOUBLE) >= 0.0099999999999999
  |  hasNullableGenerateChild: true
  |  cardinality: 1009517
  |  column statistics:
  |  * date-->[1.6687008E9  1.67E+09  0 4  24.822695035460992] ESTIMATE
  |  * case-->[-Infinity  Infinity 0 1  1.0] UNKNOWN
  |  * sum-->[0.0099999999999999  1.71E+10  0 16  7422.0] ESTIMATE
  |
  104:EXCHANGE
     cardinality: 1922890
```

在Aggregate节点在计算group by的时候丢列导致crash。

Bug Trace: 0801-0806

```
PLAN FRAGMENT 2(F00)

  Input Partition: RANDOM
  OutPut Partition: UNPARTITIONED
  OutPut Exchange Id: 04

  3:AGGREGATE (update serialize)
  |  STREAMING
  |  aggregate: count[([6: id_string, BIGINT, true]); args: BIGINT; result: BIGINT; args nullable: true; result nullab
  |  group by: [39: expr, VARCHAR, false]
  |  cardinality: 0
  |  column statistics:
  |  * expr-->[-Infinity, Infinity, 0.0, 1.0, 1.0] ESTIMATE
  |  * count-->[0.0, 0.1, 0.0, 8.0, 1.0] ESTIMATE
  |
  2:AGGREGATE (update serialize)
  |  group by: [6: id_string, BIGINT, true]
  |  cardinality: 0
  |  column statistics:
  |  * id_string-->[3.9082874048525E14, 9.2232177797603656E18, 0.0, 8.0, 438375.0] ESTIMATE
  |
  1:Project
  |  output columns:
  |  6 <-> [6: id_string, BIGINT, true]
  |  39 <-> '2022-08-02 00:00:00'
  |  cardinality: 0
  |  column statistics:
  |  * id_string-->[3.9082874048525E14, 9.2232177797603656E18, 0.0, 8.0, 438375.0] ESTIMATE
  |  * expr-->[-Infinity, Infinity, 0.0, 1.0, 1.0] ESTIMATE
  |
```

下面的表达式搞错了导致上面的project在算列的时候丢列了

```
|  repeat: repeat 0 lines [[103, 104]]
|
19:Project
|  <slot 87> : 87: dc_user_id
|  <slot 103> : 103: unnest
|  <slot 104> : datediff(CAST(split(regexp_extract(102: array_join, concat('(([\d]+){=}(', CAST(103: unnest AS VARCHAR), ')){1}'), 1), '=')[2] AS DATETIME), 103: unnest)
|  <slot 107> : if(datediff(CAST(split(regexp_extract(101: array_join, concat('((', CAST(103: unnest AS VARCHAR), '){=}([\d]+)'), 1), '=')[1] AS DATETIME), 103: unnest) = clone(clone(datediff(CAST(split(regexp_extrac(array_join(99:
|  common expressions:
|  <slot 114> : CAST(103: unnest AS VARCHAR)
|  <slot 115> : concat('(([\d]+){=}(', 114: cast, '))')
|
18:TableValueFunction
|
17:Project
|  <slot 87> : 87: dc_user_id
|  <slot 97> : 97: array_agg
|  <slot 101> : array_join(98: array_agg, ',')
|  <slot 102> : array_join(99: array_agg, ',')
|
16:AGGREGATE (merge finalize)
|  output: array_agg(97: array_agg), array_agg(98: array_agg), array_agg(99: array_agg), min(100: min)
|  group by: 87: dc_user_id
|  having: 100: min = 0
|
15:EXCHANGE

LAN FRAGMENT 3
OUTPUT EXPRS:
PARTITION: HASH_PARTITIONED: 87: dc_user_id, 85: date_trunc, 86: expr
```

列类型不匹配:

我们的union中的类型必须是一样的(nullable可以不一致)，否则就会出现结果不对或者是crash

```
0:UNION
|  child exprs:
|      [2, VARCHAR, false] | [4, INT, true]
|      [170, VARCHAR, true] | [158, VARCHAR, true]
|  pass-through-operands: all
|  hasNullableGenerateChild: true
|  limit: 10
|  cardinality: 10
```

有些时候我们是可以直接通过plan看出来哪些列是有问题的，但是在很多case下由于plan过于复杂, 可能先从BE侧发现是在哪里crash了，然后找到发生 crash 的plan node已经crash相关的列就可以更快

的定位到发生错误的地方

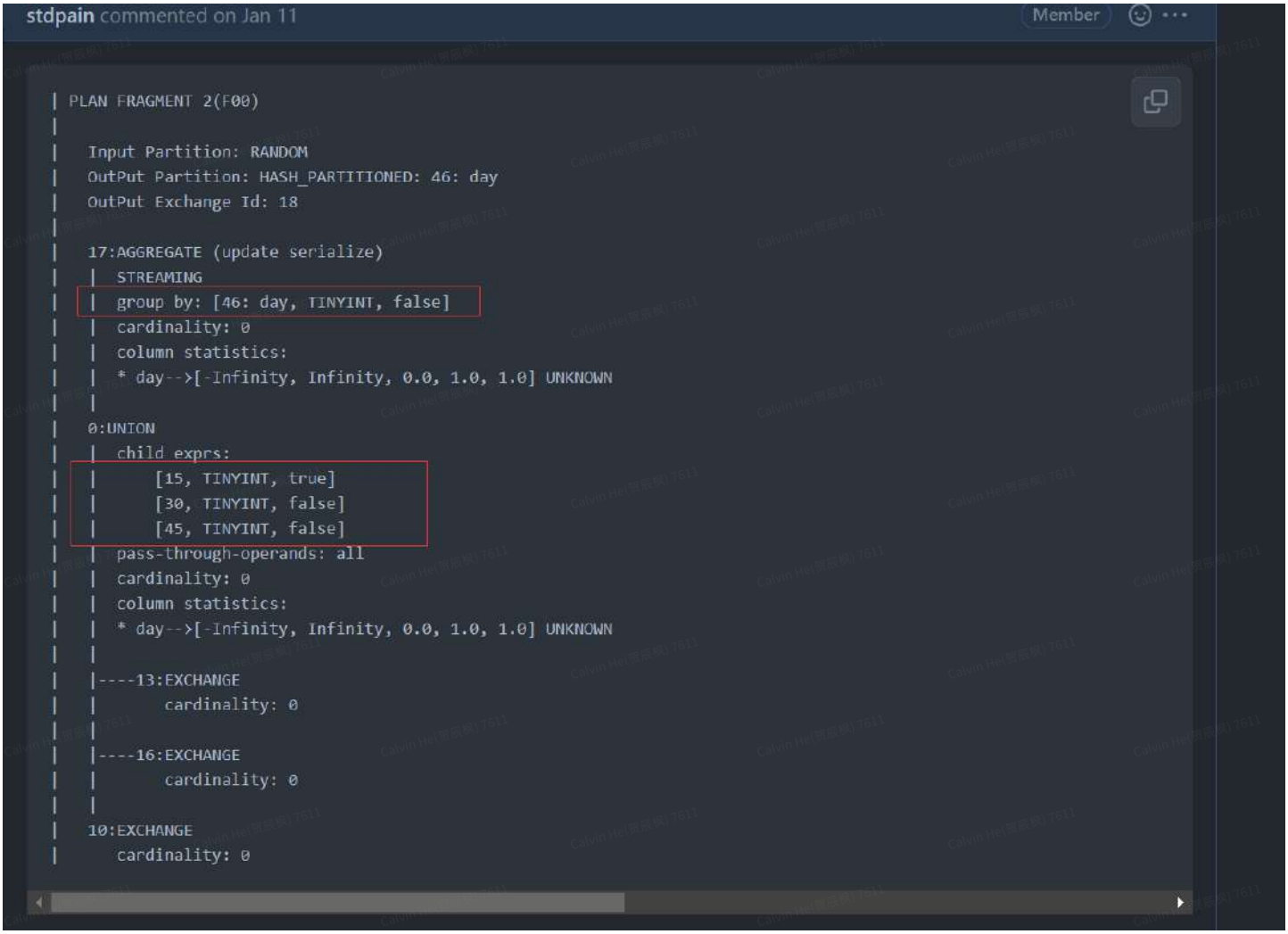
Nullable 属性给的不对

https://github.com/StarRocks/starrocks/issues/6713

https://github.com/StarRocks/starrocks/issues/2761

在这个case中 union 可能向上吐nullable column，但是 aggregate 节点认为这一列是非nullable的，这就会导致我们的BE crash掉

我们nullable属性一般都是nullable兼容非nullable的，比如group by列是一个nullable的，但是下面传递给上面的列是非nullable的，这种情况下是一个合法的plan。



一般定位到是具体的plan问题，我们就可以通过query dump把建表和查询语句dump出来，然后把这个query重放出来。

最小复现case的重要性:

理论上来讲，plan出错都是可以通过观察Plan来确定真的是Plan的问题。但是实际上由于用户的SQL可能并不是人写的，有可能是机器生成的 (比如是某些BI工具生成的)。之前也有用户提供过几十万行的explain，理论上的确是可以用人眼看出来，但是实际上面对这种case基本上是没办法在几十万行中精准的确定这个plan是否是正确的。一方面我们开发了一些plan的校验工具，但是总会有一些漏网之鱼

(优化器总是认为出的plan是正确的，即使是校验了也大概率校验不出来)。所以如果能把一个bug的范围逐渐缩小，排查复现的成本就会低很多

2. 通过 coredump定位

coredump是我们最基本的一个排查方式，他会把进程使用的内存都dump出来，我们可以根据coredump中的信息比较准确的把进程crash的时候的运行状态全都拿到。但是coredump毕竟也是一个具体的现象，如果只是简单的nullable不匹配或者是列的类型不匹配，我们拿到的coredump基本上是可以直接定位的。但是如果是某些数组越界引起的，或者是内存写乱了(use-after-free/double free) 这会导致我们crash的位置跟实际发生错误的位置会差很远。

打开coredump的方式: (需要确定好crash的盘的空间是否足够)

```
ulimit -c unlimited
sudo echo core > /proc/sys/kernel/core_pattern
```

查看coredump的方式:

```
gdb lib/starrocks_be core.xxxx
```

检查是否真的可以生成coredump (强制crash 也可以用于排查死锁/hang)

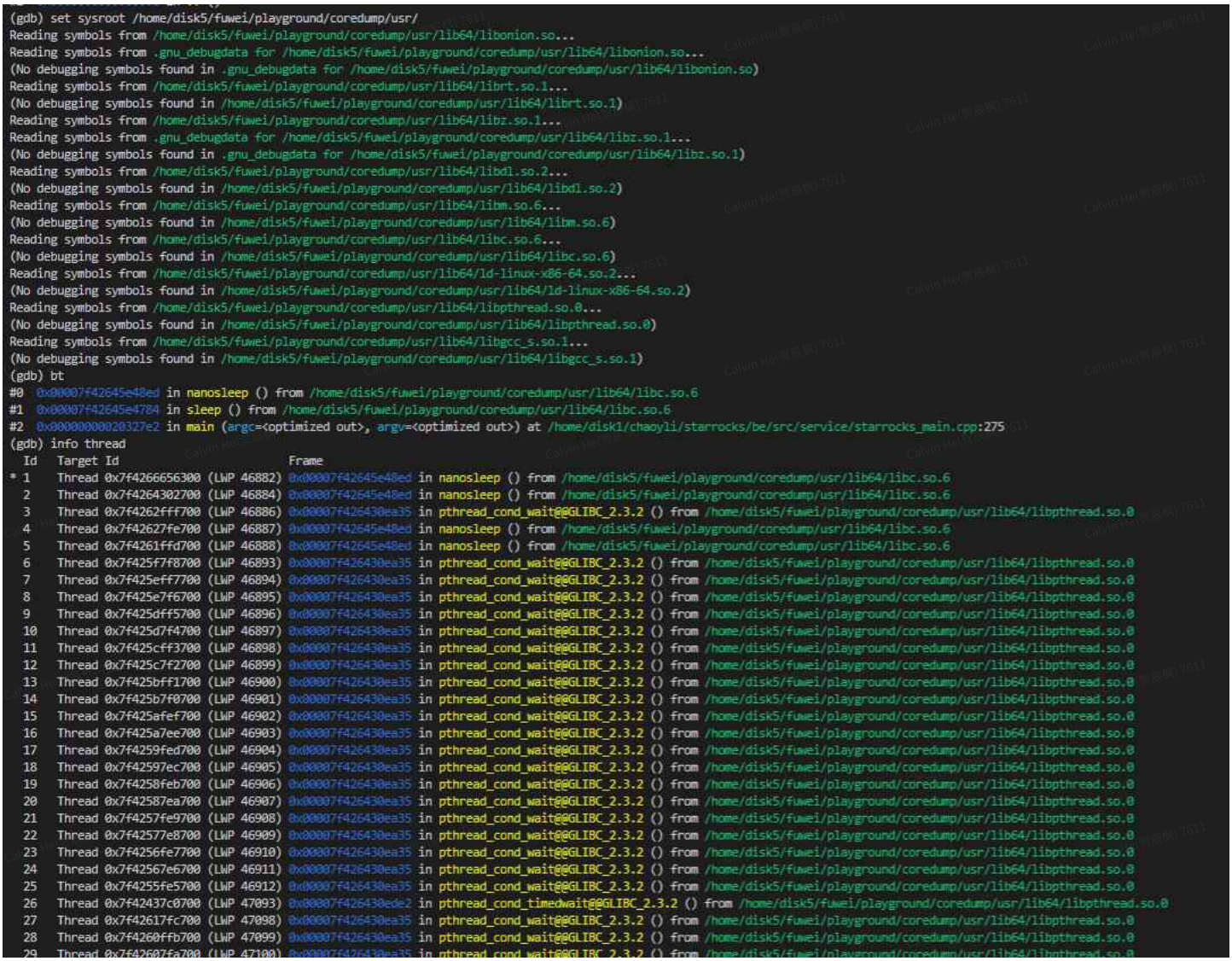
```
kill -11 $PID (-11 -6 都可以)
```

一些注意事项:

binary 需要跟产生core的保持一致，否则打开全都是问号 (不同机器编译的同一个版本也可能是不一致的)

core需要保持完整，有的用户把还在core的文件直接发过来了，导致打开之后全都是问号

某些用户环境下libc跟我们自己的libc版本差距较大，主要现象就是代码行号对不上或者都是问号，需要下载用户的 /lib64，然后手动设置 sysroot 才能正确加载符号



我们在加载core之后首先需要明确crash的原因，已经定位crash SQL。crash 主要是两个原因

- 访问了非法内存，一般收到的信号是 SIGSEV
  - 这类一般都是空指针/类型不匹配/use-after-free
- 某些没有catch住的exception导致或者是CHECK没过，一般是 SIGABORT
  - 可能是类型不匹配/overflow/bad_alloc
- SIGILL 非法指令，一般都是缺少AVX2

https://starrocks.atlassian.net/browse/SR-14210

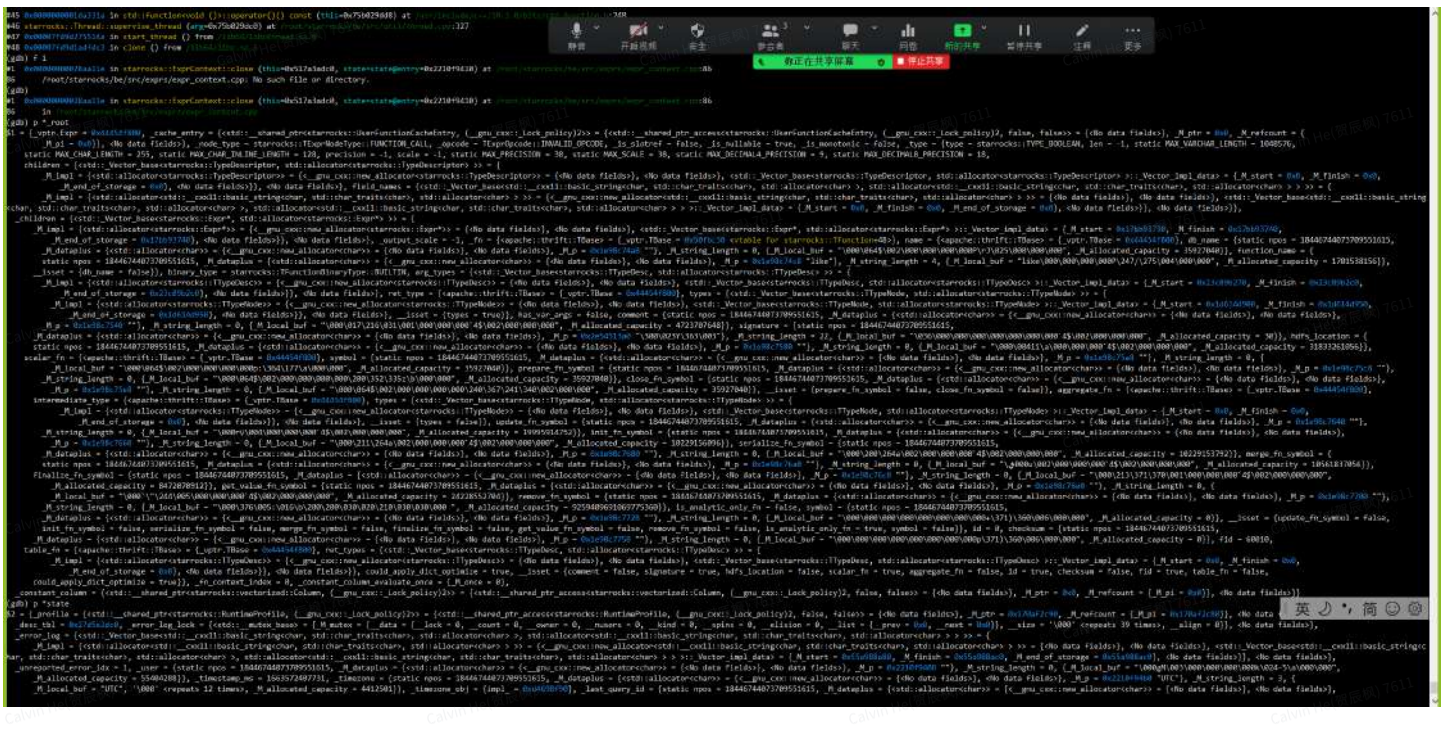
```
terminate called after throwing an instance of 'std::length_error'
  what():  vector::_M_range_insert
*** Aborted at 1669773332 (unix time) try "date -d @1669773332" if you are using
PC: @     0x7f03bb3a1387 __GI_raise
*** SIGABRT (@0x3e80002e85c) received by PID 190556 (TID 0x7f0357f3b700) from PI
    @           0x354a282 google::(anonymous namespace)::FailureSignalHandler()
    @     0x7f03bc06c630 (unknown)
    @     0x7f03bb3a1387 __GI_raise
    @     0x7f03bb3a2a78 __GI_abort
```

```
10      @          0x15fa229 _ZN9__gnu_cxx27__verbose_terminate_handlerEv.cold
11      @          0x4fad726 __cxxabiv1::__terminate()
12      @          0x4fad791 std::terminate()
13      @          0x4fad8e4 __cxa_throw
14      @          0x15fbe0d std::__throw_length_error()
15      @          0x176c102 std::vector<>::_M_range_insert<>()
16      @          0x1766874 starrocks::vectorized::BinaryColumn::append()
17      @          0x2103306 starrocks::vectorized::NullableColumn::append()
18      @          0x20f01b2 starrocks::vectorized::ArrayColumn::append()
19      @          0x2103306 starrocks::vectorized::NullableColumn::append()
20      @          0x23a4c72 starrocks::vectorized::CrossJoinNode::_copy_joined_rows
21      @          0x23a6129 starrocks::vectorized::CrossJoinNode::get_next_internal
22      @          0x23a6632 _ZNSt17_Function_handlerIFN9starrocks6StatusEPNS0_12Run
23      @          0x211e0b4 starrocks::ExecNode::get_next_big_chunk()
24      @          0x23a3998 starrocks::vectorized::CrossJoinNode::get_next()
25      @          0x23d8d41 starrocks::vectorized::ProjectNode::get_next()
26      @          0x1cdbcda starrocks::PlanFragmentExecutor::_get_next_internal_vec
27      @          0x1cdd615 starrocks::PlanFragmentExecutor::_open_internal_vectori
28      @          0x1cde1e7 starrocks::PlanFragmentExecutor::open()
29      @          0x1c79a22 starrocks::FragmentExecSta
```

use-after-free 可能的表现:

use-after-free 在crash的时候可以通过coredump来看这个指针指向的内容，正常情况下所有带函数表的指针中vtable都是会有具体的值的，但是已经free的对象中vtable一般都是一个乱的值

Tips 我们可以直接打印某个指针的vtable p *((void***)p)



1. 跟数据强相关的case:

在某些极端的控制流下，我们的框架会出现一些边界条件不清晰的问题。这也会导致BE Crash，通常情况下是很难复现这类问题的，因为它跟我们的数据控制流是有关的，需要特别精心的去构造这类case才能复现出来。排查思路也是从coredump中的信息来反向推理我们的框架执行状态。

https://github.com/StarRocks/starrocks/issues/13264

https://github.com/StarRocks/starrocks/issues/1494

3. 通过 Mini dump 定位

Mini dump是作为一个实验性质的排查手段，我们的代码中也集成了minidump，相对于coredump来说，minidump在dump的过程中，资源消耗的更少，dump出来的文件也会更小一些。我们可以通过mini2core 等工具把minidump 转换成coredump。但是minidump中dump出来的信息会少很多，只能查看有限的部分栈上变量，堆中的变量一般也是看不到的，也看不到vtable。虽然能提供的信息有限，但是有的时候如果真的没办法获取coredump，minidump也是一个可选的排查手段

> Minidump的使用方式

4. Gdb attach

Gdb attach 是通过gdb attach到一个BE上，然后通过一些断点来查看线程的运行状态。通常是用于排查死锁或者是查询hang住。这个可能导致某些请求卡在这个BE上，一般线上也很少用这个方法，有的时候debug环境下可以采取这个方法

```
gdb lib/starrocks $PID
```

BugTrace: Daily Bug 定位复盘

5. pstack

Pstack 原理跟gdb相似，功能基本上等同于 thread apply all bt. 可以把我们的BE所有线程的运行状态都dump出来，通常是用来判断我们的线程是卡在哪里的，或者是用于排查死锁等信息。

pstack通常是使用用户自带的gdb进行dump的，如果用户自带的gdb版本比较低，pstack打出来的变量也是不准确的，这个时候可能需要升级gdb或者是通过环境变量来控制选用的gdb.

例如在centos中我们为了获取准确的stack信息，我们就需要手动的设置 GDB=xxxx/gdb-10.2/bin/gdb

/bin/pstack 的内容:

```
backtrace="bt"
if test -d /proc/$1/task ; then
    # Newer kernel; has a task/ directory.
    if test `/bin/ls /proc/$1/task | /usr/bin/wc -l` -gt 1 2>/dev/null ; then
        backtrace="thread apply all bt"
    fi
```

```
elif test -f /proc/$1/maps ; then
    # Older kernel; go by it loading libpthread.
    if /bin/grep -e libpthread /proc/$1/maps > /dev/null 2>&1 ; then
        backtrace="thread apply all bt"
    fi
fi

GDB=${GDB:-gdb}
```

6. 通过 ASAN 排查内存问题:

https://github.com/google/sanitizers/wiki/AddressSanitizer

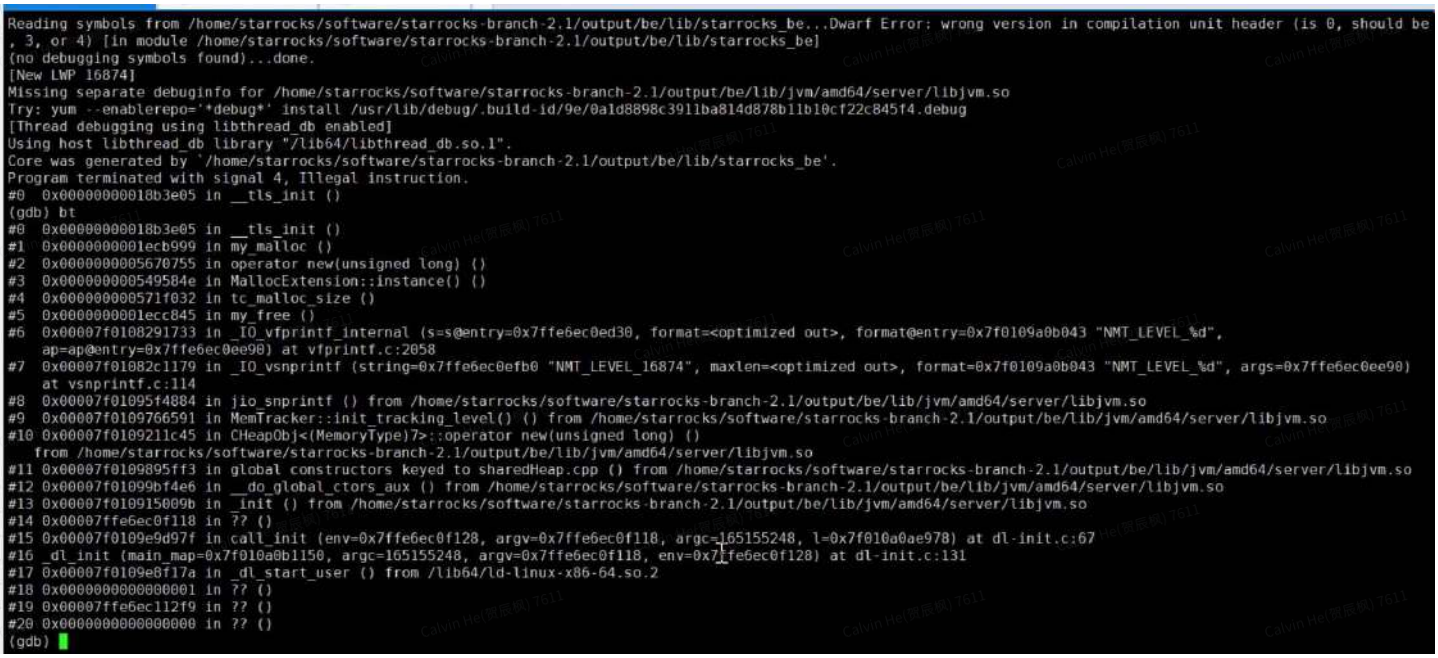
我们的BE集成了ASAN，当我们遇到随机Crash的时候就可以尝试用ASAN找到出问题的地方:

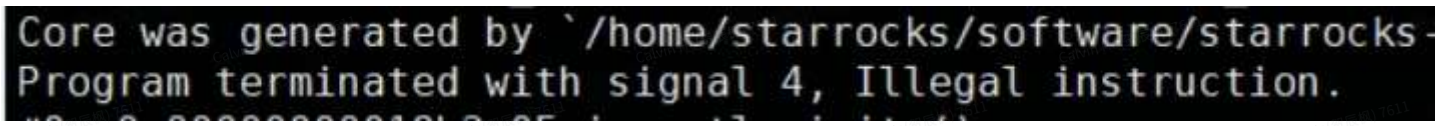
一般内存有这几个问题:

2. heap-buffer-overflow
3. use-after-free
4. stack-buffer-overflow

环境引起的crash

硬件缺少指令





我们官网发行包都是带avx2的，有些用户的机器上没有avx2，所以会报这个错误。但是有的环境上上在be.out中打不出来带SIGILL的log，但是通过 coredump 是可以看出来的。

glibc不兼容引起的crash:

有的时候由于用户自己部署的环境版本比较低，导致没办法使用StarRocks。我们StarRocks编译要求glibc的版本至少是 2.17。太低或者是太高可能都会有问题。

```
1 /lib64/libc.so.6: version `GLIBC_2.14' not found in Pre-compiled binary
```

这类问题都是属于glibc的兼容问题。最保险的做法就是手动的编译一个合适的glibc版本，然后通过修改启动脚本去手动的加载指定的glibc。

https://github.com/StarRocks/starrocks/issues/192